ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте на тему:

Мобильное приложение по чтению документов со стола

Выполнил студент:

группы #БПМИ223, 2 курса Бондаренко Марк Александрович

Принял руководитель проекта:

Днестрян Андрей Игоревич
Приглашенный преподаватель
Факультет компьютерных наук НИУ ВШЭ

Москва 2024

Содержание

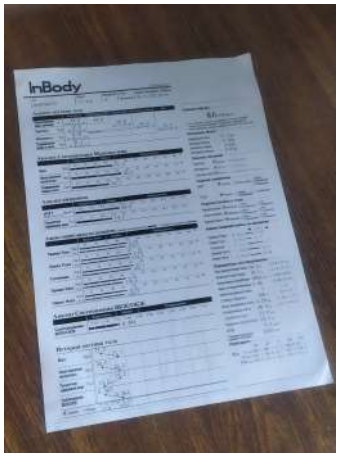


Рис. 1.1: Пример входного изображения.

Рис. 1.2: Пример обработаного изображения.

1 Аннотация

Данная курсовая работа посвящена разработке андроид приложения - SnapDoc. SnapDoc редназначен для создания электронной копии докумена, по его фотографии. Был реализован следующий функционал у приложения: пользователь имеет возможность создавать папки со страницами документа, на странице папки есть возможность загрузить изображение из галереи или сделать фото (1.1), далее алгоритм пытается найти на изображении углы документа и предлагает пользователю поправить их в случае неточностей, наконец изображение обрезается, нормализуется и добавляется к остальным страницам документа. Пользователь может собрать pdf документ со страницами изображения представленными в папке по нажатию на кнопку и сохранить его на устройство. (1.2).

Ключевые слова

Компьютерное зрение, мобильное приложение, Flutter, детектор углов, нормализатор, SnapDoc, изображение, виджет

2 Введение

В современном информационном обществе, где цифровизация и автоматизация процессов становятся все более важными, разработка мобильных приложений, способных упростить повседневные задачи пользователей, играет значительную роль. В рамках данной курсовой работы было поставлено задание разработать мобильное приложение, способное создать электронную копию документа по фотографии.

Целью данного проекта является разработка алгоритма обработки изображения осно-

ванного на существующих и исключение недостатков присущих старым решениям в контексте решения конкретной задачи: отцифровка документа по фотографии. А также разработка мобильного приложения, которое будет являться удобным интерфейсом по его использовнию.

В рамках данной работы будут рассмотрены основные этапы разработки приложения, а также проведен анализ его функциональности и потенциальных преимуществ перед существующими аналогами. Особое внимание будет уделено технологиям компьютерного зрения и обработки изображений, которые используются для реализации функционала распознавания углов документа на фотографии.

3 Обзор существующих решений

В ходе проведенного анализа был осуществлен обзор существующих мобильных приложений, выполняющих задачи, аналогичные функционалу разработанного мною продукта. Исследование показало, что большинство приложений в данной категории обладают схожим набором функций, предлагая пользователю стандартные инструменты для работы с документами. Мое приложение также вписывается в данный функциональный ряд, предоставляя базовый комплект необходимых опций. В свете этого обстоятельства, особый интерес представляет сопоставление не столько самих приложений, сколько алгоритмов распознавания углов документов, лежащих в их основе. Поэтому рассмотрим основные подходы к их реализации.

3.1 Детектор углов Харриса [1]

Этот метод использует алгоритм Харриса для обнаружения углов на изображении. Он ищет точки, где есть значительные изменения яркости в разных направлениях, что обычно соответствует углам объектов на изображении. Чуствителен к освещению и шуму на изображении, что может привести к ложным обнаружениям углов. Не эффективен для изображений с низким разрешением или большим количеством деталей, что яляется серьезной проблемой, так как содержание документа может быть абсолютно произвольным: включать в себя различные иллюстрации, иметь не шаблонный дизайн с большим количеством деталей.

3.2 Детектор FAST [2]

Этот метод быстро и эффективно обнаруживает углы на изображении, используя локальные интенсивные точки. Он может быть хорошим выбором для распознования обьектов

в реальном времени из за своей высокой по сравнению с другими методами скоростью. Однако метод является в разы менее точным по сравнению с остальными решениями, особенно на сложных изображениях. К тому же не всегда обнаруживает углы с высокой степенью поворота или масштабирования.

3.3 Методы машинного обучения [3]

Методы машинного обучения на данный момент являются самыми точными. С помощью сверточных нейросетей удается добиться очень точных результатов, но это при достаточной сложности сети. Да при увелечении количества слоев точность вырастает до поразительных результатов, однако вместе с ней возрастает и время работы. Обученная сверточная нейросеть с большим количеством слоев будет работать крайне долго, что является главной проблемой этого подхода. Можно использовать легкие модели, однако обучить такую модель до действительно хороших результатов - крайне трудная задача. Поэтому в текущей работе методы машинного обучения планируется использовать только для решения подзадач или вообще не использовать.

3.4 Преобразование Хафа [4]

Этот метод используется для поиска прямых линий на изображении, но его также можно применить для поиска углов. После обнаружения линий можно искать их пересечения, которые могут быть углами документа. Является вычислительно затратным особенно на больших изображениях, а также неточен на фотографиях с низким контрастом.

4 Этапы разработки

Задачу можно разбить на 3 крупные подзадачи: детектор углов, нормализатор и разработка интерфейса мобильного приложения с интеграцией детектора углов и нормализатора. Детектор углов и нормализатор - функции, предоставляемые библиотекой SnapDocLib написанной на языке c++.

4.1 Детектор углов

Детектор углов представляет собой функцию, которая принимает на вход путь к изображению формата jpg и указатель на массив целочисленных координат углов, в которые

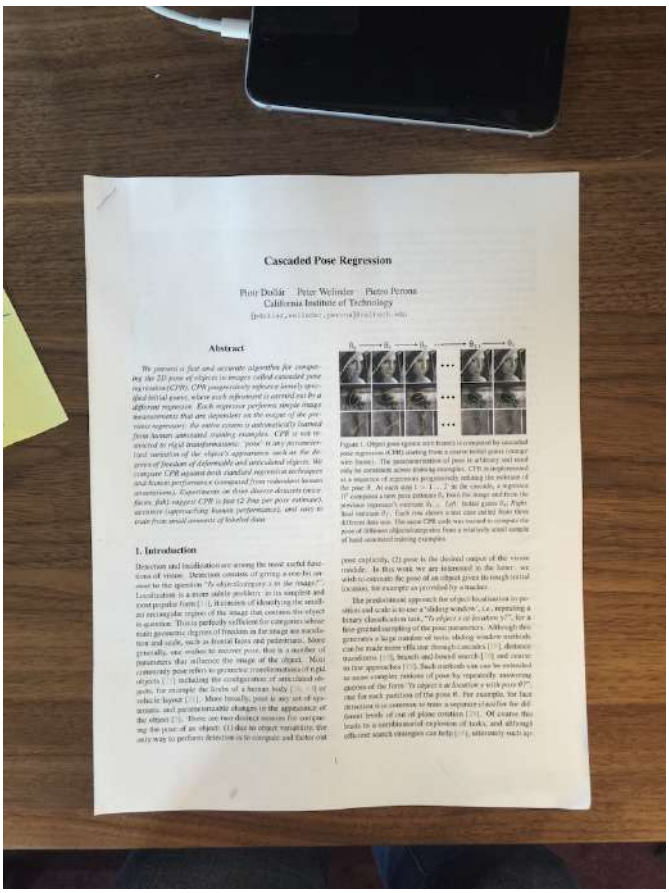


Рис. 4.1: Входное изображение

будет записан результат работы, а возвращает код равный 0, если функция отработала корректно, и не ноль, если произошла какая-то ошибка (4.2).

```
int32_t DetectCorners(const char* path, int32_t* corners);
```

Рис. 4.2: Сигнатура функции DetectCorners

За основу алгоритма решено взять LSD [5]. Такой подход ранее не использовался в решении данной задачи, хотя обладает преимуществами по сравнению с вышеперечисленными. Все это будет подробнее обсуждаться далее (4.1).

4.1.1 Чтение jpg и выделение наиболее информативного канала

Для декодирования JPG используется библиотека OpenCV. В моем коде объявлены такие классы как `SingleChanneledImage`, `DoubleImage`, `BoolImage`, которые по сути являются удобной оберткой над двумерными массивами `int32_t`, `double` и `bool` соответственно, со всеми методами необходимыми для работы с изображением (4.3). `SingleChanneledImage` хранит в себе значения интенсивности канала пикселя, которое варируется от 0 до 255. Изображение считывается в 4 объекта `SingleChanneledImage`: красный, зеленый, синий и канал

```cpp
template<typename T>
class Image {
public:
    Image();
    Image(int32_t width, int32_t height);
    int32_t Width() const;
    int32_t Height() const;
    void SetPixel(int32_t y, int32_t x, T pixel);
    T GetPixel(int32_t y, int32_t x) const;
    void Clear();

private:
    int32_t height_ = 0;
    int32_t width_ = 0;
    std::vector<T> data_;
};

using SingleChanneledImage = Image<int32_t>;
using BoolImage = Image<bool>;
using DoubleImage = Image<double>;
```

Рис. 4.3: Класс изображения

яркости. Первые 3 получаются напрямую из RGB изображения, а вот канал яркости вычисляется по формуле (1) [6]:

$$lightness = 0.299 \cdot red + 0.587 \cdot green + 0.114 \cdot blue \tag{1}$$

Далее происхоид отбор канала. Прежде всего отмечу, что документ от окружающего фона может отличаться не только по яркости, но и по цвету. Поэтому, как бы нам ни хотелось сократить пространство поиска, нельзя просто избавляться от цветовых каналов в изображении. Нужно выбрать тот, который наиболее подходит для дальнейшей работы. Для этого вычисляются гистограммы 4-х каналов. Далее из гистограмм вычисляются средние значения (2) и дисперсии (3) каналов. Выбирается канал с максимальной дисперсией. Высокая дисперсия присуща изображениям с высоким контрастом, а это свойство, которое значительно облегчает работу любого алгоритма распознавания обьектов.

$$\mu = \frac{1}{256} \sum_{i=0}^{255} h(i) \tag{2}$$

$$\sigma^2 = \frac{1}{256} \sum_{i=0}^{255} (h(i) - \mu)^2 \tag{3}$$

4.1.2 Предобработка изображения

Перед тем как приступить к дальнейшему анализу, изображение необходимо подготовить так, чтобы улучшить и ускорить работу алгоритмов выделения границ. Сначала происходит этап масштабирования - исходное изображение уменьшается в $ScaleFactor$ раз для уменьшения объема данных, что ускоряет последующую обработку. Значение $ScaleFactor$ выбирается как наименьшее целое число, удовлетворяющее условию (4):

$$\frac{\text{width}}{ScaleFactor} \times \frac{\text{height}}{ScaleFactor} < MAX\_SIZE, \tag{4}$$

где width и height — ширина и высота исходного изображения соответственно, а MAX_SIZE — заданная константа, которая в данной работе равна 50 000. Уменьшение размера изображения осуществляется методом свертки с усредняющим фильтром (4.4). На графиках (4.5) и (4.6) представлена зависимость среднего времени работы DetectCorners от MAX_SIZE, а также MMD от MAX_SIZE, на тестировочном датасете. MMD - Median Max Deviation. Max Deviation выхода функции расчитывается как максимум по всем углам расстояния от реальной точки нахождения угла, до точки которую обнаружил алгоритм. Соответственно MMD это медианное значание Max Deviation на тестировочном датасете. Как видно из графиков данный порог был выбран именно таким, потому что меньший дает уже крупную ошибку по сравнению с вообще не масштабированным изображением.

После масштабирования применяется Гауссово размытие для улучшения дальнейшего выделения границ. Используется фильтр с фиксированным ядром размером 5×5, что оптимально для изображений уменьшенного размера. Гауссово размытие помогает сгладить шум и детали изображения, улучшая тем самым процесс выделения границ (4.7). Это достигается за счет применения Гауссовой функции:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{5}$$

где G — значение Гауссовой функции в точке с координатами (x, y), а σ — стандартное отклонение распределения.

4.1.3 Нахождение отрезков (LSD)

Line Segment Detector подробно описан в статье. Алгоритм был реализован на языке C++, и представляет собой функцию со следующей сигнатурой (4.8): `LineSegment` - структура отрезка. Она хранит в себе координаты концов отрезка: (x1, y1) и (x2, y2), длину отрезка:

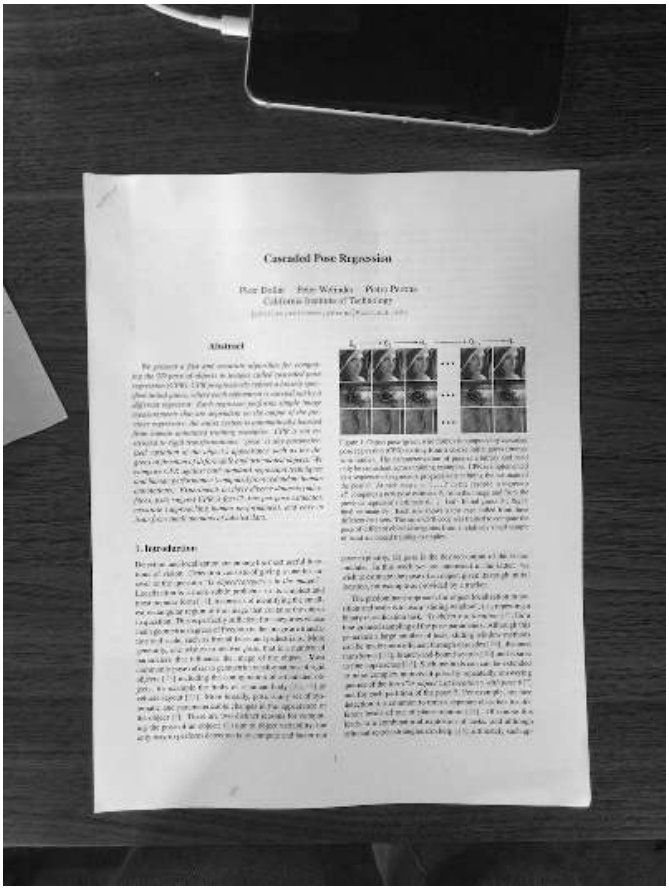


Рис. 4.4: Масштабированное изображение

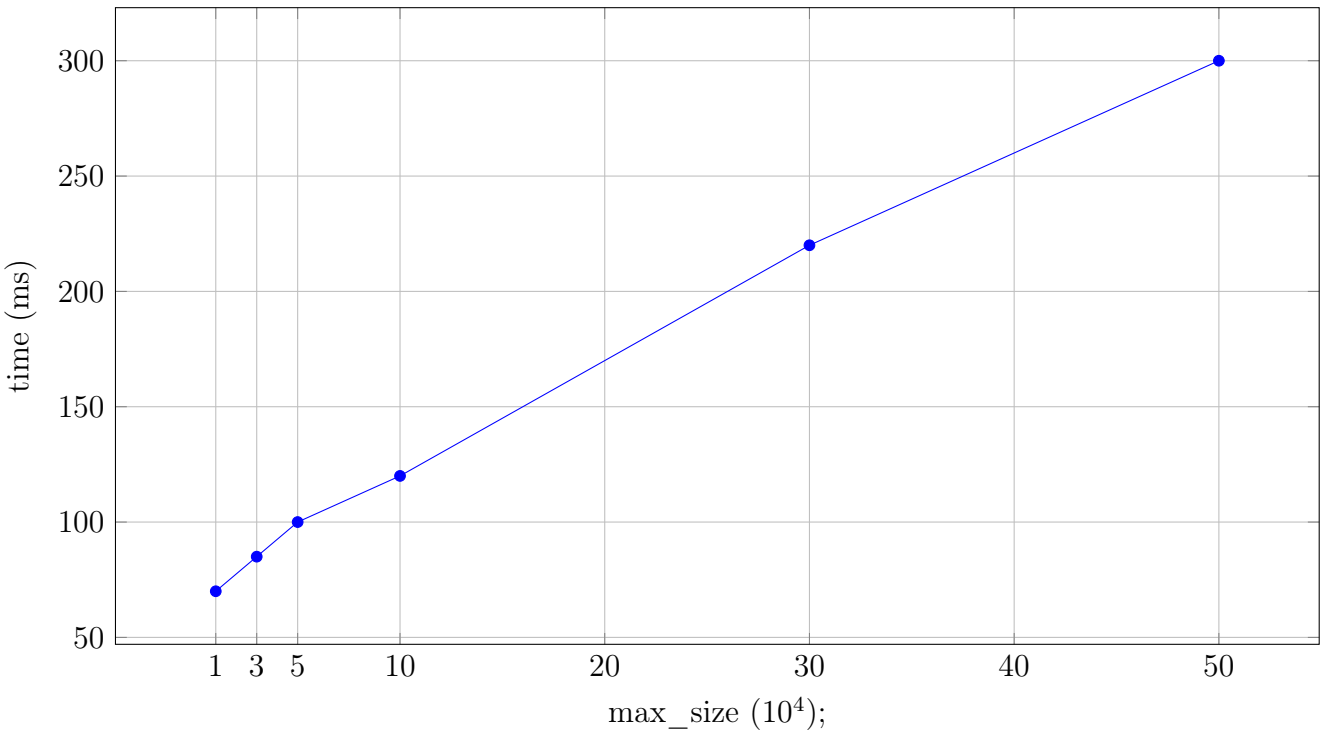


Рис. 4.5: Зависимость времени выполения DetectCorners от MAX_SIZE

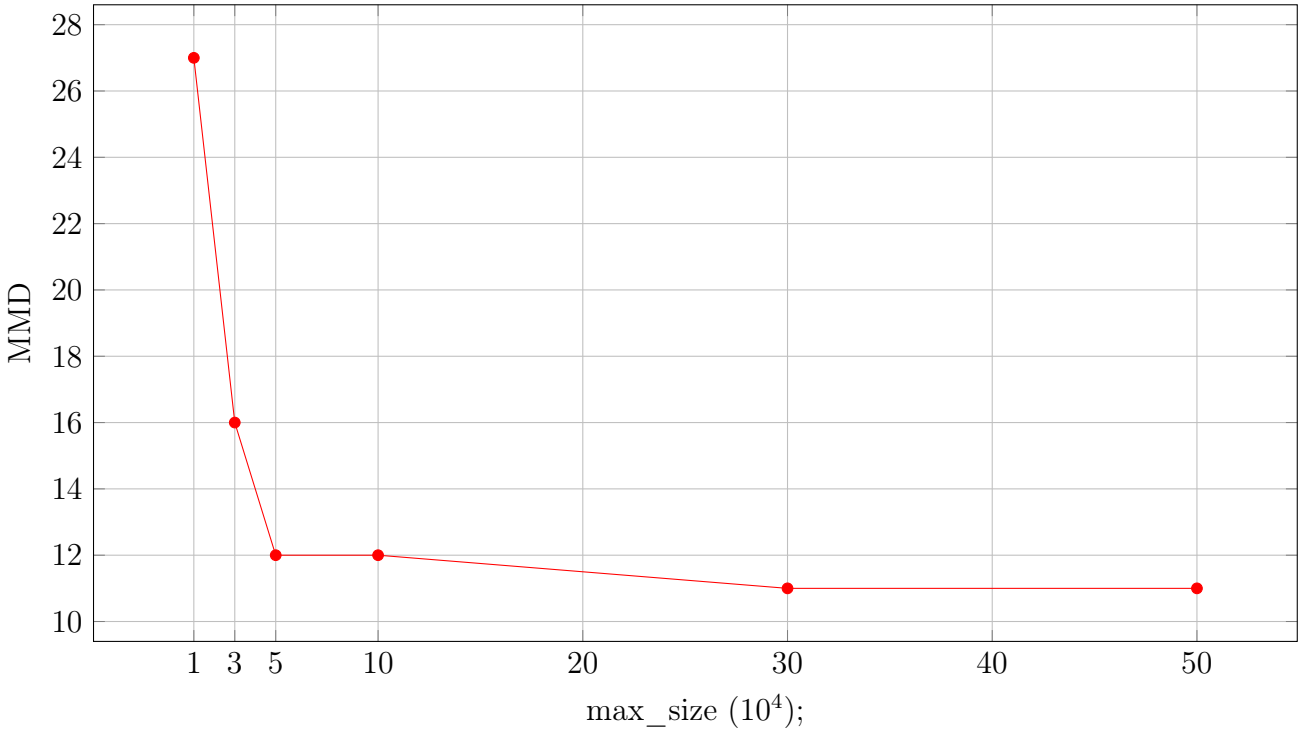


Рис. 4.6: Зависимость MMD от MAX_SIZE

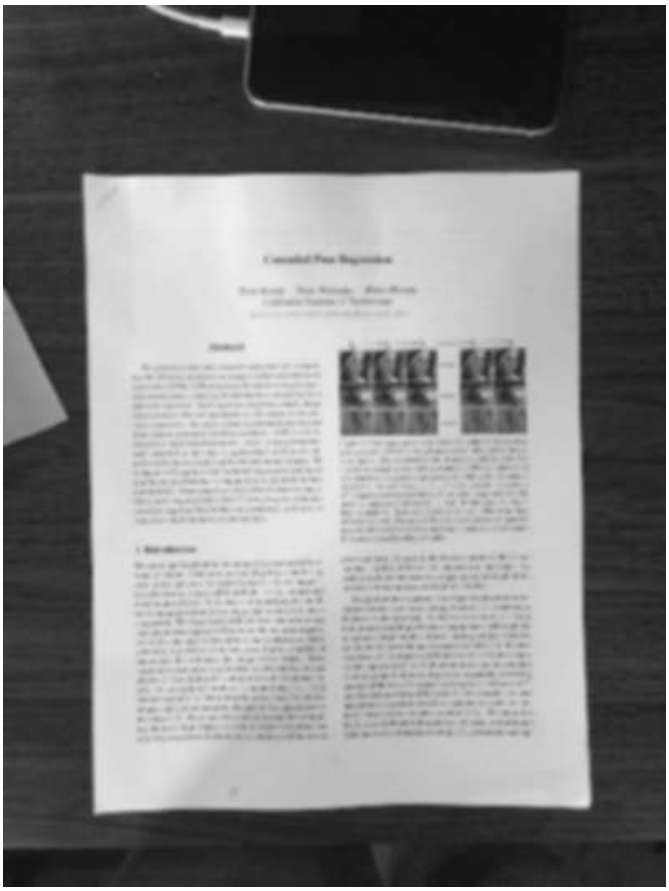


Рис. 4.7: Размытое изображение

```cpp
struct LineSegment {
    int32_t x1 = 0;
    int32_t y1 = 0;
    int32_t x2 = 0;
    int32_t y2 = 0;
    int32_t length = 0;
    double angle = 0.0;
};

std::vector<LineSegment>
    DetectLineSegments(const SingleChanneledImage& img);
```

Рис. 4.8: Структура LineSegment и сигнатура функции DetectLineSegments

length, а также угол его наклона в радианах принадлежащий отрезку от -π до π: angle. Визуализировано на изображении (4.9).

4.1.4 Нахождение прямых

После выделения отрезков производится формирование прямых линий. Каждая прямая представлена в коде структурой `Line` (см. Рисунок 4.10).

В полярных координатах прямая задается углом θ (поле `angle`) и расстоянием r от начала координат до ближайшей точки прямой (поле `r`). Поле `segments` представляет список отрезков на прямой, а поле `weight` — сумму длин этих отрезков. Для segments верно: любые 2 отрезка из segments не имеют общих точек, отрезки сортированы по координате левых (нижних) концов. Функция `GetLines` (4.11), отвечает за группировку отрезков в прямые.

Группировка отрезков основана на их угловой выравненности. Функция `IsAligned` сравнивает углы двух отрезков и возвращает `true`, если они равны с учетом заданной погрешности `angle_tolerance`. Углы считаются равными, если их разность по модулю 2π меньше или равна `angle_tolerance`:

$$\text{IsAligned}(\alpha, \beta) = \min\left(|\alpha - \beta|, 2\pi - |\alpha - \beta|\right) \leq \text{angle\_tolerance}, \tag{6}$$

где `angle_tolerance` равна $\frac{\pi}{8}$ и используется как в алгоритме LSD, так и при определении принадлежности отрезка к прямой.

Формирование линии происходит жадно: отрезки сортируются под длине, и начиная с самых длинных алгоритм пытается расширить линию.

Процедура добавления отрезков в прямую включает корректировку параметров `Line` в соответствии с добавляемым отрезком. Параметры r и θ обновляются следующим образом

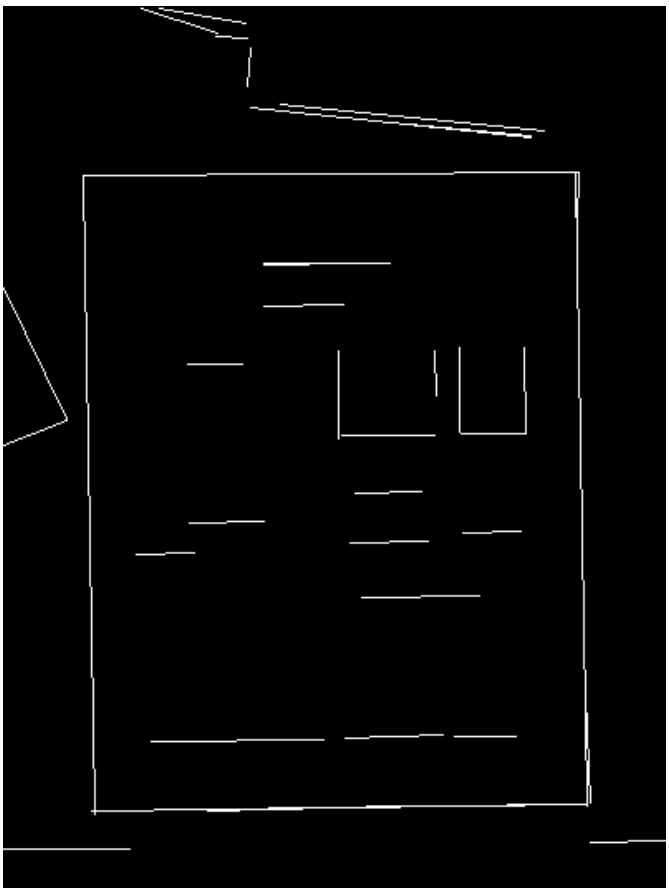


Рис. 4.9: Отрисованные отрезки полученные из DetectLineSegments

```
struct Point {
    int32_t y = 0;
    int32_t x = 0;
};

struct Line {
    double angle = 0.0;
    double r = 0.0;
    double weight = 0.0;
    std::vector<std::pair<Point, Point>> segments;
};
```

Рис. 4.10: Структуры Point и Line

```
std::vector<Line>
    GetLines(std::vector<LineSegment>& line_segments);
```

Рис. 4.11: Сигнатура функции GetLines

(формулы 7 и 8):

$$r_{\text{new}} = \frac{\text{Line.weight} \cdot \text{Line.r} + \text{Segment.length} \cdot \text{Segment.r}}{\text{Line.weight} + \text{Segment.length}}, \tag{7}$$

$$\theta_{\text{new}} = \arctan\left(\frac{\text{Line.weight} \cdot \sin(\text{Line.angle}) + \text{Segment.length} \cdot \sin(\text{Segment.angle})}{\text{Line.weight} \cdot \cos(\text{Line.angle}) + \text{Segment.length} \cdot \cos(\text{Segment.angle})}\right), \quad (8)$$

К `weight` добавляется длина добавляемого отрезка. Концы всех отрезков проецируются на итоговую прямую, а пересекающиеся и вложенные отрезки объединяются, формируя один непрерывный сегмент. Когда прямая полностью сформирована и соответствует всем критериям, она помещается в объект `BoundedLineHeap`. Этот объект представляет собой кучу, хранящую ограниченное количество элементов и поддерживающую `AMOUNT_OF_LINES_TO_PROCESS` прямых с наибольшим значением `weight`. Так как `AMOUNT_OF_LINES_TO_PROCESS` является константой, равной 25, вставка в `BoundedLineHeap` осуществляется с постоянной временной сложностью O(1). Это позволяет избежать необходимости полной сортировки массива прямых, значительно оптимизируя процесс обработки данных. В конце алгоритма из кучи достаются линии и складываются в вектор, для формирования возвращаемого значения (4.12).

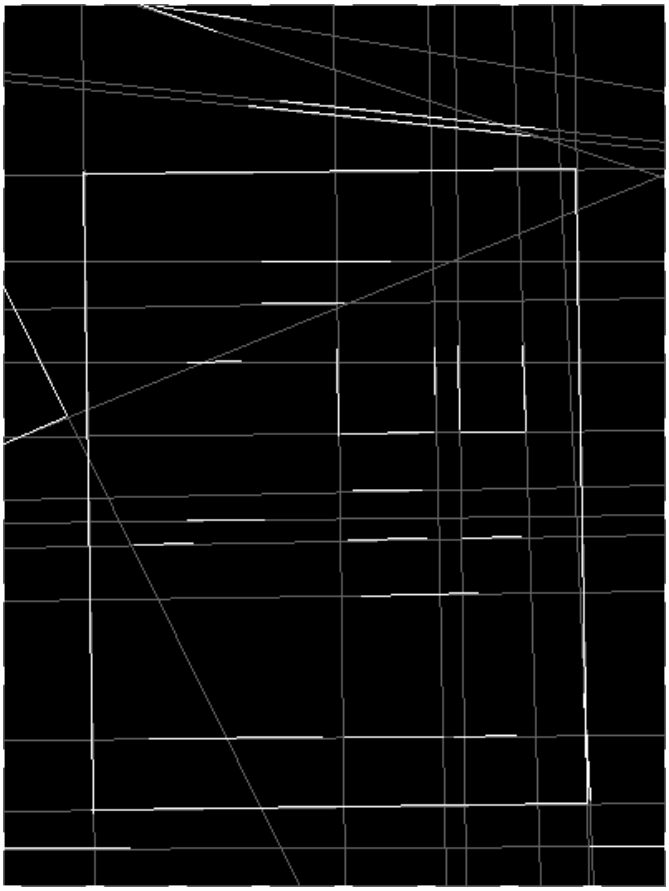


Рис. 4.12: Отрисованные прямые и отрезки на них. Прямые - серые, отрезки - белые.

4.1.5 Добавление опциональных линий

После формирования основного списка прямых линий на изображении, в него дополнительно включаются четыре прямые, расположенные вдоль периметра изображения. Эти линии ориентированы горизонтально и вертикально и предназначены для случаев, когда не удалось выделить достаточно линий для формирования правильного ответа. Так же такой подход дает гарантию на ответ по умолчанию у алгоритма. Ответом по умолчанию будут углы, соответствующие углам изображения

Вертикальные линии добавляются таким образом, чтобы отношение weight к height было приблизительно равно обратному значению OPTIONAL_SEGMENT_PERIOD, и аналогично для горизонтальных линий, где weight относится к width:

$$\frac{\text{weight}}{\text{height}} \approx \frac{1}{\text{OPTIONAL\_SEGMENT\_PERIOD}}, \quad \text{для вертикальных линий,} \tag{9}$$

$$\frac{\text{weight}}{\text{width}} \approx \frac{1}{\text{OPTIONAL\_SEGMENT\_PERIOD}}, \quad \text{для горизонтальных линий.} \tag{10}$$

Эффект достигается путем добавления отрезков с фиксированной длиной OPTIONAL_SEGMENT_SIZE и интервалами OPTIONAL_SEGMENT_PERIOD вдоль каждой стороны изображения. В реализации эти параметры имеют следующие значения: OPTIONAL_SEGMENT_PERIOD = 5 и OPTIONAL_SEGMENT_SIZE = 10 (4.12). Такой трюк обеспечивает алгоритму возможность в случае правильного определения лишь части углов, получить некоторый результат в котором будут фигурировать эти углы, а также точки расположенные на периметре изображения (4.13). Замечу, что если алгоритм вообще не найдет подходящих кандидатов, то он вернет углы изображения, как значение по умолчанию.

4.1.6 Нахождение углов документа

После формирования списка прямых, следует определить четыре прямые, которые с наибольшей вероятностью соответствуют сторонам документа на изображении. Пересекая эти прямые, находятся углы документа.

Для решения задачи были разработаны структура `Candidate` и класс `CandidateDiscoverer` (4.14).

Структура `Candidate` включает в себя следующие ключевые параметры: - **corners**: Вектор точек, представляющих предполагаемые углы документа. - **perimetr_density**: Плотность периметра, измеряющая длину покрытого отрезками участка периметра по отно-

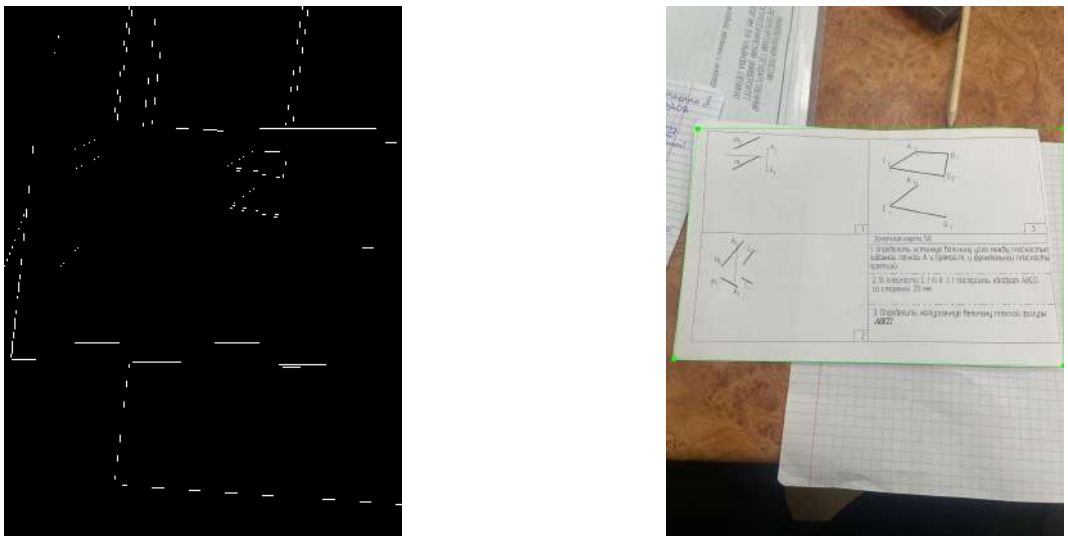


Рис. 4.13: Слева изображены распознанные границы документа, а справа результат детекции. Как видно правая сторона не выделилась так как лежала на белом фоне. Алгоритм зацепился за 3 реальные стороны документа и левую опциональную линию и за счет этого корректно определил 3 из 4 углов.

```
struct Candidate {
    std::vector<Point> corners;
    double perimetr_density;
    double area_percentage;
    double sides_parallelism;
    Candidate();
};

class CandidateDiscoverer {
public:
    CandidateDiscoverer(const std::vector<Line>& lines,
                        int32_t img_width, int32_t img_height);

    Candidate GetCandidate(int32_t line1, int32_t line2,
                           int32_t line3, int32_t line4);

private:
    Matrix line_intersections_;
    std::vector<double> line_angles_;
    int32_t img_width_;
    int32_t img_height_;
    std::vector<Point> points_;
    std::vector<std::unordered_map<int32_t, double>>
                                        prefix_weights_;
};
```

Рис. 4.14: Описание структуры `Candidate` и класса `CandidateDiscoverer`

шению к полной длине периметра, образованной точками из `corners`.

$$\text{perimetr\_density} = \frac{\text{covered perimeter length}}{\text{total perimeter length}} \tag{11}$$

- **area_percentage**: Отношение площади четырёхугольника, образованного точками из `corners`, к площади всего изображения. Этот параметр показывает, какая часть изображения занята предполагаемым документом.

$$\text{area\_percentage} = \frac{\text{quadrilateral area}}{\text{image area}} \tag{12}$$

- **sides_parallelism**: Мера того, насколько четырехугольник приближен к форме прямоугольника, с максимальным значением 1, где 1 означает идеальный прямоугольник. Этот параметр основан на сравнении параллельности противоположных сторон.

$$\text{sides\_parallelism} = \frac{|\cos(l1.\text{angle} - l3.\text{angle})| + |\cos(l2.\text{angle} - l4.\text{angle})|}{2} \tag{13}$$

Класс `CandidateDiscoverer` используется для поиска наилучшего кандидата среди возможных четверок прямых. Основные поля класса включают: - **line_intersections_**: Матрица пересечений линий, где элемент матрицы содержит индекс точки пересечения соответствующих линий, хранящийся в **points_**, или -1, если точка пересечения прямых не существует или не принадлежит прямоугольнику [-`AIRBAG`, img_width_ + `AIRBAG`] x [-`AIRBAG`, img_height_ + `AIRBAG`]. `AIRBAG` - констаната, которая в моем коде равна 30. Это зона в 30 пикселей вокруг изображения в которой алгоритм все еще обрабатывает углы (4.15). То есть если пользователь неровно сфоткал документ и уголок слегка вышел за пределы изображения, то засчет добавления "подушки безопасности"алгоритм отработает корректно и загонит угол лежащий за пределами изображения внутрь изображения. - **line_angles_**: Углы прямых. - **prefix_weights_**: Префиксные длины покрытого участка для каждой прямой, позволяющие определять длину покрытого участка на отрезке между двумя точками пересечения на прямой за время $O(1)$. То есть prefix_weights_[i][j] равно длине покрытого участка на прямой lines[i] от [-∞, -∞] до points_[j].

Для нахождения лучшего кандидата из списка прямых строится `CandidateDiscoverer`, запускается перебор всех возможных четверок прямых, и по каждой строится объект структуры `Candidate` с помощью метода `GetCandidate`. Высчитывается функция правдоподобия для каждого кандидата:

Параметры весов для функции правдоподобия в моём коде определены следующим образом: $PD\_K = 50.0$, $AP\_K = 20.0$, $SP\_K = 20.0$. Выбирается кандидат с наибольшим значением функции правдоподобия.

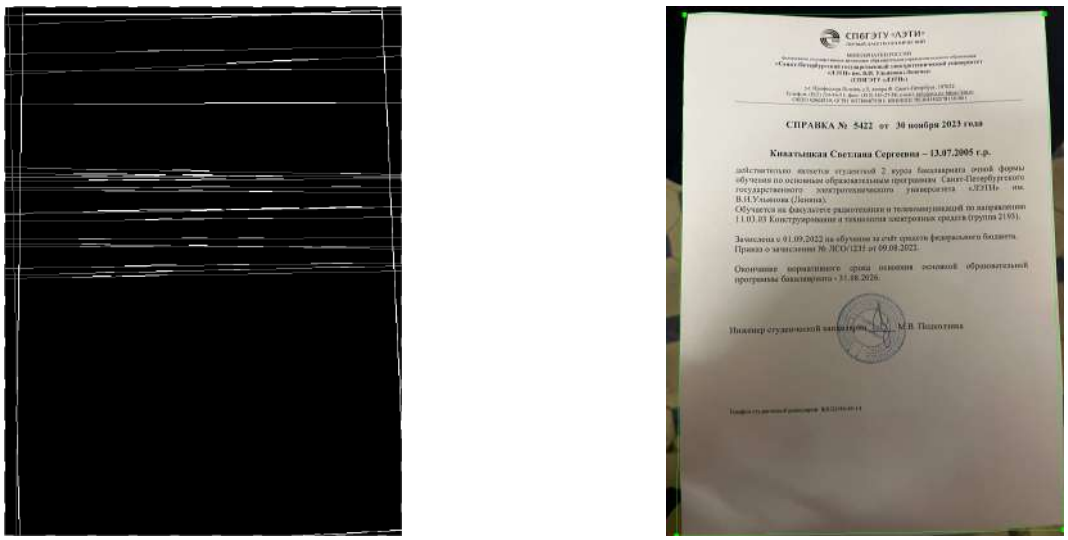


Рис. 4.15: Слева изображены линии распознаные на изображении, как видно не все точки пересечения сторон документа находятся на изображении. Тем не менее засчет трюка с AIRBAG получаем результат, который изображен справа, соответствующий ожиданиям пользователя.

```
double Likelihood(const Candidate& candidate) {
    return PD_K * candidate.perimetr_density
           + AP_K * candidate.area_percentage
           + SP_K * candidate.sides_parallelism;
}
```

Рис. 4.16: Функция оценки правдоподобия кандидата

4.1.7 Упорядочивание углов документа

Когда обнаружены четыре угла документа, следующим шагом является их упорядочивание для последовательной обработки и анализа. Важно, чтобы углы были упорядочены таким образом, чтобы первый угол в списке соответствовал левому верхнему углу документа, а остальные следовали в порядке обхода по часовой стрелке.

Выбор начального угла: Первым шагом является выбор начальной точки, которая будет использоваться как опорный угол для дальнейшего упорядочивания. Эта точка выбирается как угол с наименьшим евклидовым расстоянием до начала координат, что почти наверное соответствует действительности.

Упорядочивание оставшихся углов: После выбора начальной точки, оставшиеся три угла упорядочиваются с использованием векторного произведения для определения направления обхода. Векторное произведение двух векторов $\vec{AB}$ и $\vec{AC}$ (где A — начальная точка, B и C — две другие точки) позволяет определить, находится ли точка C по часовой

стрелке относительно точки B при взгляде из точки A:

$$\vec{AB} \times \vec{AC} = (x_B - x_A)(y_C - y_A) - (y_B - y_A)(x_C - x_A). \tag{14}$$

Знак векторного произведения указывает направление: - Если произведение положительно, точка C находится по часовой стрелке относительно точки B. - Если произведение отрицательно, точка C находится против часовой стрелки.

Используя эту логику, упорядочиваются три оставшиеся точки так, чтобы они следовали в порядке обхода по часовой стрелке относительно выбранного начального угла.

Этот метод упорядочивания гарантирует, что углы будут обработаны в порядке, который соответствует предполагаемой ориентации документа на изображении, что критично для работы нормализатора (4.17).

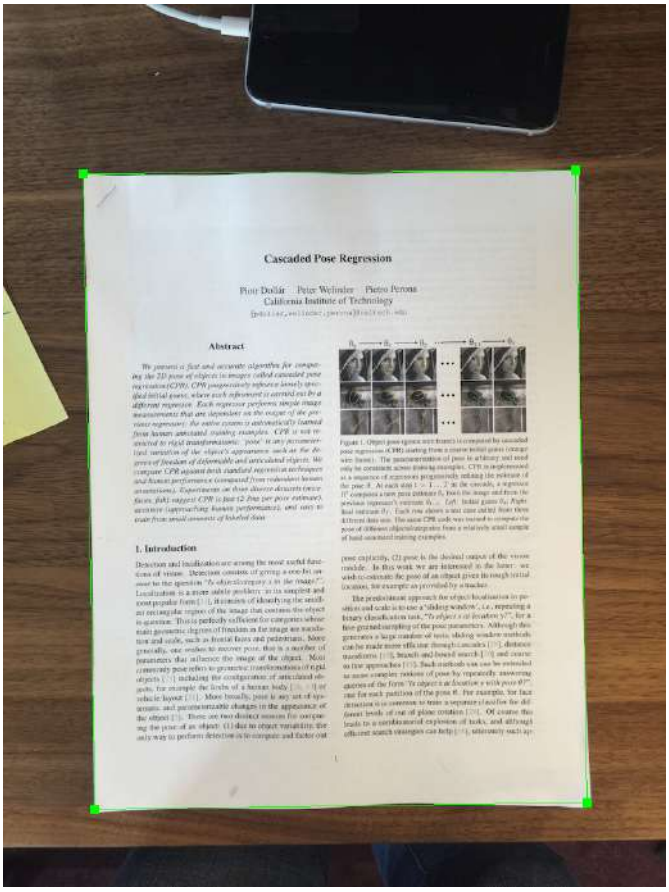


Рис. 4.17: Визуализация найденных углов.

4.1.8 Оценка работы алгоритма

Алгоритм распознавания углов документов был тщательно протестирован на датасете, содержащем 41 реальное изображение различных документов. Испытания проводились на

изображениях, сделанных разнообразными камерами в различных условиях освещения и на разных фонах, что позволило оценить универсальность и адаптивность алгоритма.

Результаты тестирования показали, что в 31 случае алгоритм успешно идентифицировал все четыре угла документа. В пяти случаях были распознаны некоторые углы, но не все, и в оставшихся пяти случаях алгоритм не смог корректно определить ни одного угла.

Более глубокий анализ ошибочных распознаваний, особенно на этапе обнаружения отрезков с использованием алгоритма LSD, указывает на то, что низкий контраст в определённых областях изображений стал основной причиной проблем. Все пять изображений, где алгоритм не смог распознать ни одного угла, объединяет общий фактор: документы размещены на белой или светлой поверхности, что затрудняет выделение контрастных границ.

Эта проблема является общей для многих алгоритмов распознавания границ и в рамках текущей работы найти решение, которое позволило бы справиться с низким контрастом, не удалось. Тем не менее, алгоритм показал впечатляющие результаты при распознавании документов, расположенных под различными углами в перспективе, он устойчив к наличию множества деталей, нечувствителен к шумам и работает довольно быстро.

Временные показатели работы алгоритма на тестовом датасете показали, что среднее время обнаружения углов, затрачиваемое функцией `DetectCorners`, составляет 100 миллисекунд. Максимально время обработки не превышает 250 миллисекунд. Отмечается, что значительная часть времени (порядка 50%) уходит на начальные этапы обработки: чтение изображения, выделение канала и его масштабирование, что связано с работой с изображениями в полном размере и соответствующим объёмом данных.

4.2 Нормализатор

Функция Normalize принимает на вход путь к исходному изображению, путь к результирующему изображению и массив из 8 целых чисел, соответствующий координатам углов документа. Возвращает функция код возврата, такой же как и у функции DetectCorners (4.18). Цель функции — преобразовать искаженное изображение в прямоугольное, исправляя перспективные искажения.

```
int32_t Normalize(const char* input,
                  const char* ouput,
                  int32_t* corners);
```

Рис. 4.18: Сигнатура функции Normalize

4.2.1 Определение длин сторон результирующего изображения

В процессе работы функции определения углов были получены следующие координаты четырёх точек: (x_1, y_1), (x_2, y_2), (x_3, y_3), и (x_4, y_4). Обозначим стороны четырёхугольника как l_1, l_2, l_3, и l_4.

Для определения центра масс четырёхугольника, вычислим его координаты:

$$x_c = \frac{x_1 + x_2 + x_3 + x_4}{4}, \quad y_c = \frac{y_1 + y_2 + y_3 + y_4}{4}. \tag{15}$$

Точка пересечения диагоналей четырёхугольника обозначена как (x_d, y_d). Отклонение центра масс от точки пересечения диагоналей вычисляется как:

$$dx = |x_c - x_d|, \quad dy = |y_c - y_d|. \tag{16}$$

Итоговые размеры результирующего изображения определяются на основе средних длин противоположных сторон четырёхугольника, скорректированных на величину отклонения:

$$\text{width} = \frac{l_1 + l_3}{2} + 2 \cdot dx, \quad \text{height} = \frac{l_2 + l_4}{2} + 2 \cdot dy. \tag{17}$$

Этот подход был протестирован и сравнен с методом, который использует среднее значение длин противоположных сторон четырёхугольника без коррекции. Эксперименты показали, что описанный выше метод предоставляет значительно более точные результаты для определения итоговых размеров изображения.

4.2.2 Преобразование перспективы

Следующим шагом является коррекция перспективы, для чего применяется функциональность библиотеки OpenCV. Была разработана функция `TransformDocument` (4.19).

```
cv::Mat TransformDocument(const cv::Mat &input_image,
                          const std::vector<cv::Point2f> &corners,
                          float width, float height);
```

Рис. 4.19: Сигнатура функции `TransformDocument`

Работа функции основана на преобразовании перспективы. Используя вычисленные значения `width` и `height`, а также координаты углов документа, определённые на предыдущем этапе, формируется матрица перспективного преобразования с помощью функции `cv::getPerspectiveTransform`. Эта матрица затем используется функци-

ей `cv::warpPerspective`, которая трансформирует входное изображение и выдаёт итоговое изображение документа с исправленной перспективой. Результатом работы функции `TransformDocument` является преобразованное изображение, готовое для дальнейшей обработки.

4.2.3 Применение фильтров

В рамках постобработки изображения, после коррекции перспективы, была проведена серия операций фильтрации с целью улучшения контраста и снижения уровня шума.

Изначально изображение `transformed` преобразовывается из цветового пространства BGR в LAB с помощью функции `cv::cvtColor`. LAB цветовое пространство позволяет отдельно работать с яркостью изображения и его цветовыми компонентами, что удобно при необходимости изменения контрастности без влияния на цвета.

Далее изображение разделяется на отдельные каналы с использованием функции `cv::split`, после чего проводится локальное улучшение контраста канала яркости с помощью алгоритма CLAHE (*Contrast Limited Adaptive Histogram Equalization*). Для этого применяется функция `cv::createCLAHE`, которая улучшает видимость деталей на изображении, особенно в тех областях, где контраст выше среднего.

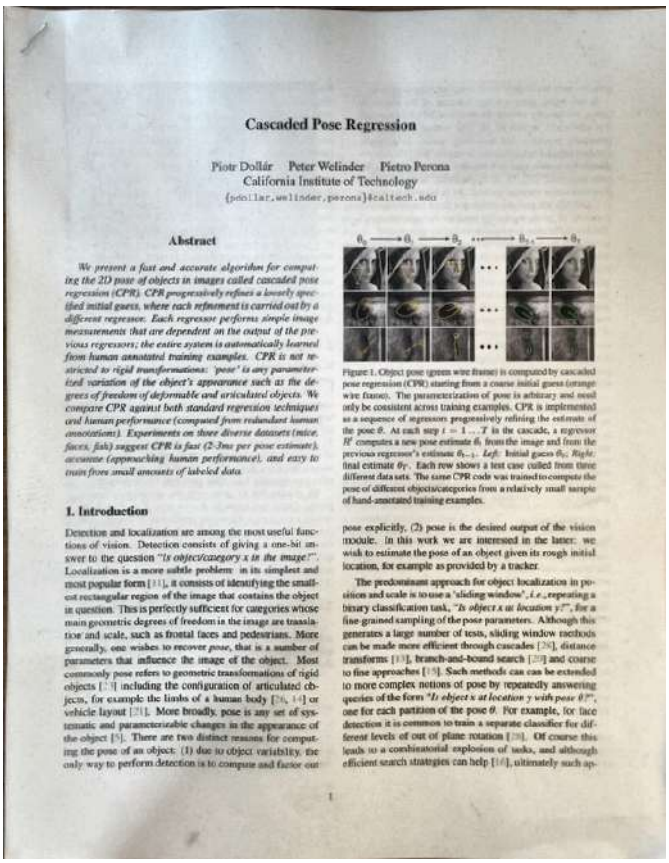
После улучшения канала яркости, каналы снова объединяются в единое изображение LAB и конвертируются обратно в цветовое пространство BGR. Это обеспечивает корректное отображение цветов на последующих этапах работы с изображением.

Для подавления шума и сглаживания изображения без потери важных границ используется билатеральный фильтр (`cv::bilateralFilter`), который эффективно удаляет шум, сохраняя при этом резкие края объектов.

Завершающим этапом является сохранение обработанного изображения на диск с помощью функции `cv::imwrite` (4.20).

4.3 Разработка мобильного приложения

Было разработано мобильное андроид приложение SnapDoc с использованием фреймворка Flutter. Динамическая библиотека `SnapDocLib`, включающая в себя реализации алгоритмов `DetectCorners` и `Normalize`, была скомпилирована для поддержки различных архитектур процессоров, таких как x86, x86_64, arm64-v8a и armeabi-v7a, с использованием инструментария Android NDK, что обеспечивает совместимость с широким спектром мобильных устройств. Интеграция нативного кода с кодом на Dart, который является основой



Cascaded Pose Regression

Piotr Dollár Peter Welinder Pietro Perona
California Institute of Technology
{pdollar,welinder,perona}@caltech.edu

Abstract

We present a fast and accurate algorithm for computing the 2D pose of objects in images called cascaded pose regression (CPR). CPR progressively refines a loosely specified initial guess, where each refinement is carried out by a different regressor. Each regressor performs simple image measurements that are dependent on the output of the previous regressors; the entire system is automatically learned from human annotated training examples. CPR is not restricted to rigid transformations: 'pose' is any parameterized variation of the object's appearance such as the degrees of freedom of deformable and articulated objects. We compare CPR against both standard regression techniques and human performance (computed from redundant human annotations). Experiments on three diverse datasets (mice, faces, fish) suggest CPR is fast (2-3ms per pose estimate), accurate (approaching human performance), and easy to train from small amounts of labeled data.

1. Introduction

Detection and localization are among the most useful functions of vision. Detection consists of giving a one-bit answer to the question "*Is object/category x in the image?*". Localization is a more subtle problem: in its simplest and most popular form [11], it consists of identifying the smallest rectangular region of the image that contains the object in question. This is perfectly sufficient for categories whose main geometric degrees of freedom in the image are translation and scale, such as frontal faces and pedestrians. More generally, one wishes to recover *pose*, that is a number of parameters that influence the image of the object. Most commonly pose refers to geometric transformations of rigid objects [23] including the configuration of articulated objects, for example the limbs of a human body [26, 14] or vehicle layout [21]. More broadly, pose is any set of systematic and parameterizable changes in the appearance of the object [5]. There are two distinct reasons for computing the pose of an object: (1) due to object variability, the only way to perform detection is to compute and factor out pose explicitly, (2) pose is the desired output of the vision module. In this work we are interested in the latter: we wish to estimate the pose of an object given its rough initial location, for example as provided by a tracker.

The predominant approach for object localization in position and scale is to use a 'sliding window', *i.e.*, repeating a binary classification task, "*Is object x at location y?*", for a fine-grained sampling of the pose parameters. Although this generates a large number of tests, sliding window methods can be made more efficient through cascades [26], distance transforms [13], branch-and-bound search [20] and coarse to fine approaches [15]. Such methods can can be extended to more complex notions of pose by repeatedly answering queries of the form "*Is object x at location y with pose θ?*", one for each partition of the pose θ. For example, for face detection it is common to train a separate classifier for different levels of out of plane rotation [28]. Of course this leads to a combinatorial explosion of tests, and although efficient search strategies can help [16], ultimately such ap-

Figure 1. Object pose (green wire frame) is computed by cascaded pose regression (CPR) starting from a coarse initial guess (orange wire frame). The parameterization of pose is arbitrary and need only be consistent across training examples. CPR is implemented as a sequence of regressors progressively refining the estimate of the pose θ. At each step $t = 1 \ldots T$ in the cascade, a regressor R^t computes a new pose estimate θ_t from the image and from the previous regressor's estimate θ_{t-1}. *Left*: Initial guess θ_0; *Right*: final estimate θ_T. Each row shows a test case culled from three different data sets. The same CPR code was trained to compute the pose of different objects/categories from a relatively small sample of hand-annotated training examples.

Рис. 4.20: Нормализованное изображение

Flutter-приложений, осуществлялась через Dart FFI (Foreign Function Interface), что позволяет приложению эффективно использовать нативную библиотеку (4.21).

4.3.1 Домашняя страница (4.22)

Домашняя страница приложения представляет из себя интерфейс для управления документами пользователя. Она включает в себя список созданных документов и кнопку для добавления новых. Каждый документ хранится на устройстве в виде папки с изображениями в формате jpg и имеет уникальное имя. Попытка создать документ с уже существующим именем приведёт к появлению уведомления о конфликте имен, благодаря чему предотвращаются ошибки в данных.

Уведомления о событиях в приложении реализованы с использованием пакета flutter_styled_toast. Доступ к файловой системе устройства осуществляется через пакет permission_handler, а работа с файлами — через модули dart:io и path_provider.

Класс `DocumentsList` (4.24), наследуемый от `ChangeNotifier`, используется для хранения списка документов (4.23). Это позволяет приложению реагировать на изменения в списке документов и обновлять интерфейс пользователя в реальном времени. Экземпляр класса `DocumentsList` функционирует как singleton, что обеспечивается с помощью пакета

```
typedef DetectCornersNative =
    ffi.Int32 Function(ffi.Pointer<Utf8>, ffi.Pointer<ffi.Int32>);
typedef DetectCornersDart =
    int Function(ffi.Pointer<Utf8>, ffi.Pointer<ffi.Int32>);

typedef NormalizeNative =
    ffi.Int32 Function(ffi.Pointer<Utf8>,
        ffi.Pointer<Utf8>, ffi.Pointer<ffi.Int32>);
typedef NormalizeDart =
    int Function(ffi.Pointer<Utf8>,
        ffi.Pointer<Utf8>, ffi.Pointer<ffi.Int32>);

class NativeLibrary {
  static late ffi.DynamicLibrary _lib;

  static void loadLibrary() {
    _lib = ffi.DynamicLibrary.open("libSnapDocLib.so");
  }

  static int detectCorners(String imagePath,
              ffi.Pointer<ffi.Int32> corners);

  static int normalize(String inputPath,
                       String outputPath,
                ffi.Pointer<ffi.Int32> corners);
}
```

Рис. 4.21: Описание интерфейса для вызова нативных функций из SnapDocLib

provider.

Класс `DocumentsList` предоставляет методы для загрузки документов из хранилища, создания новых документов, получения информации о документах, их удаления и переименования. Эти функции являются ключевыми для управления данными в приложении.

4.3.2 Страница документа (4.25)

На странице документа пользователь может управлять своими изображениями: просматривать, добавлять новые или удалять существующие.

При выборе документа открывается страница, где отображается список всех изображений, составляющих документ (4.26). Каждое изображение можно просмотреть в увеличенном размере по нажатию.

Добавление изображений: Пользователь может добавить изображения двумя способами: загрузить из галереи или сделать новое фото. Оба метода реализованы с использованием пакета image_picker. Выбранное изображение анализируется функцией поиска углов

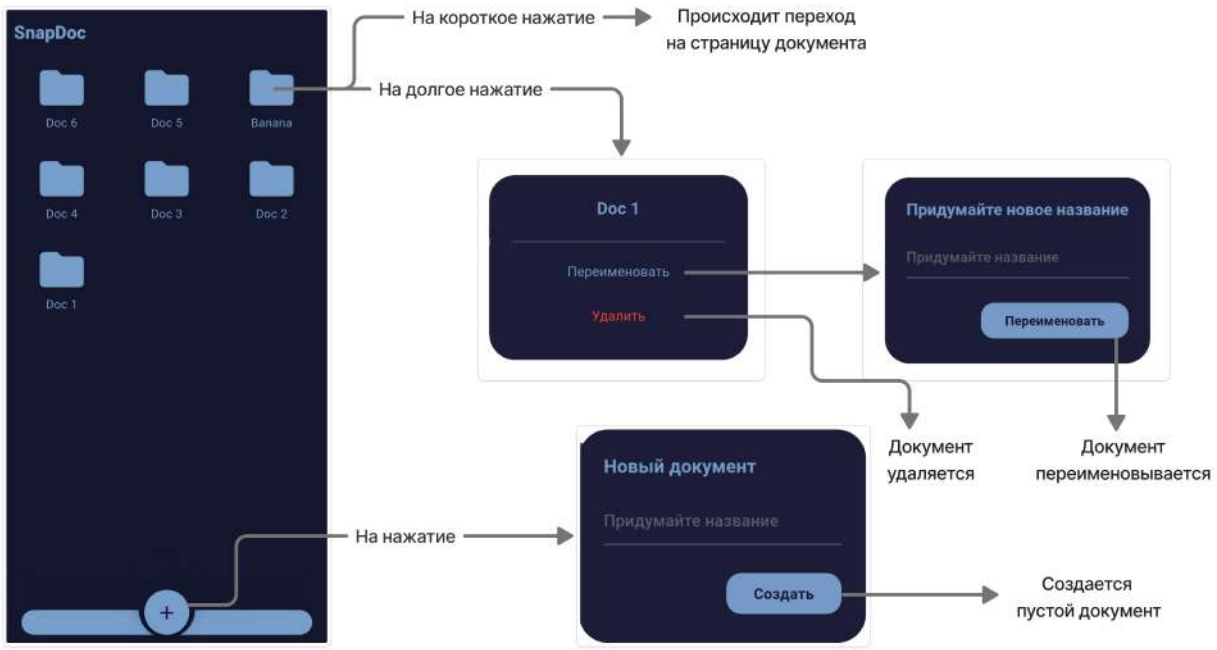


Рис. 4.22: Интерфейс домашней страницы

```
class Document {
  String name;
  Directory directory;
  Document({required this.name, required this.directory});
}
```

Рис. 4.23: Класс `Document`

```
class DocumentsList extends ChangeNotifier {
    final List<Document> _documents = [];

    Future<void> load(dynamic context) async;
    Future<void> create(String name, dynamic context) async;
    int size();
    String getName(int index);
    Document get(int index);
    Future<void> delete(int index, dynamic context) async;
    Future<void>
        rename(int index, String newName, dynamic context) async;
}
```

Рис. 4.24: Класс `DocumentsList`

(`DetectCorners`), результаты которого отображаются на изображении с возможностью коррекции положения углов пользователем.

Обработка и сохранение изображений: После коррекции углов пользова-

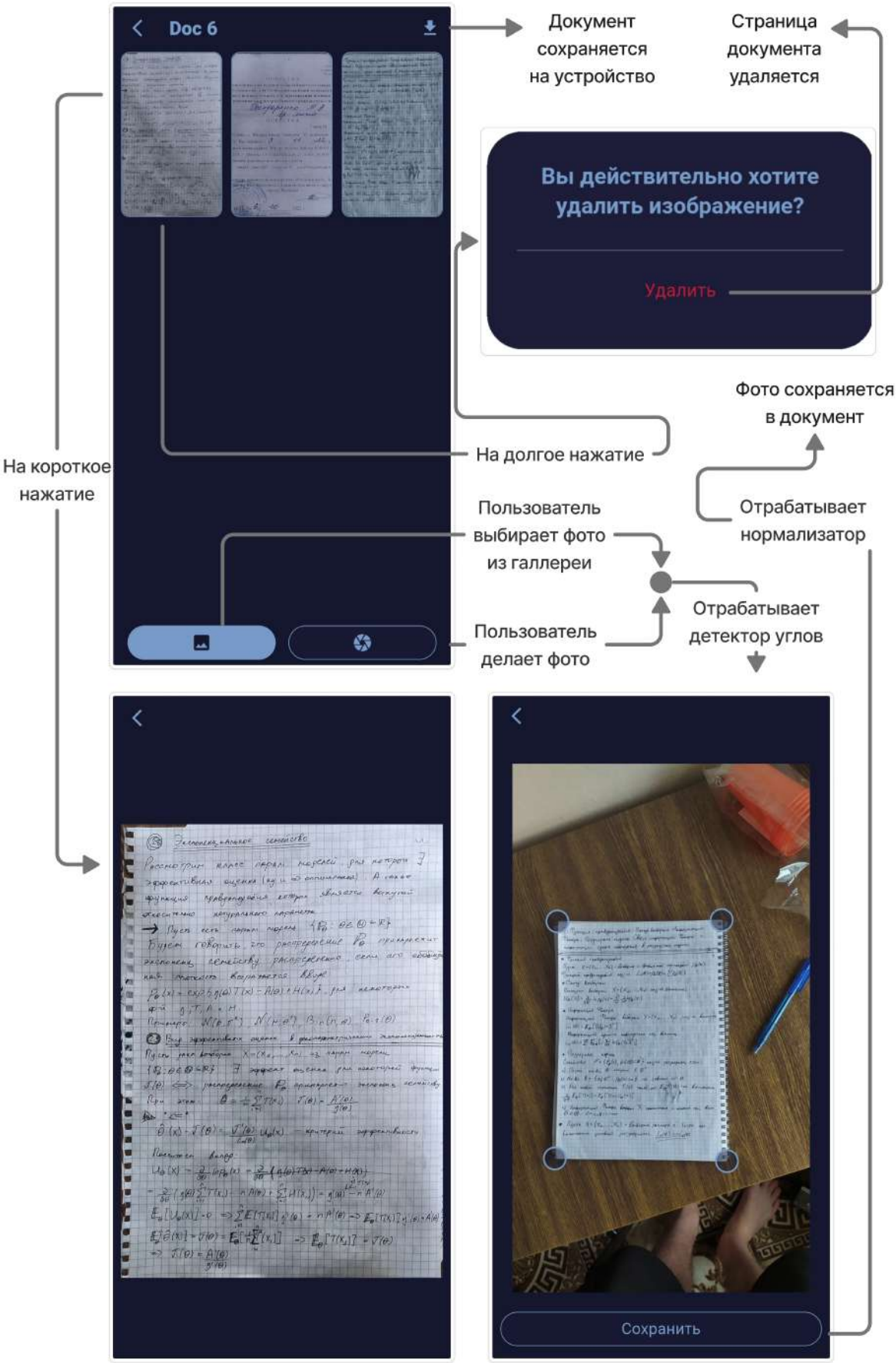


Рис. 4.25: Интерфейс страницы документа

тель может сохранить изменения, при этом изображение отправляется на нормализацию (`Normalize`), и обработанный результат записывается в папку документа. В случае ошибок в процессе любой из функций, пользователь получит соответствующее уведомление.

Создание PDF: На странице также присутствует кнопка для генерации PDF-документа из собранных изображений. Эта функция активируется только если документ содержит изображения. Сборка PDF осуществляется с помощью пакетов pdf и image, и результат сохраняется в папку `Documents` на устройстве пользователя.

```
class Page {
  File file;
  Page({required this.file});
}

class PagesList extends ChangeNotifier {
  final List<Page> _data = [];

  Future<void> load(Directory directory, dynamic context) async;

  int size();

  void clear();

  Page get(int index);

  void remove(int index);
}
```

Рис. 4.26: Классы изображения документа и списка. Логика аналогична логике с DocumentsList.

Список литературы

[1] *Harris corner detector*. URL: https://en.wikipedia.org/wiki/Harris_corner_detector (дата обр. 06.01.2024).

[2] *Features from accelerated segment test*. URL: https://en.wikipedia.org/wiki/Features_from_accelerated_segment_test (дата обр. 26.08.2023).

[3] *SPIKING HIERARCHICAL NEURAL NETWORK FOR CORNER DETECTION*. URL: https://pure.ulster.ac.uk/ws/portalfiles/portal/11286273/36824.pdf (дата обр. 01.01.2011).

[4] *Hough tranform*. URL: https://en.wikipedia.org/wiki/Hough_transform#:~:text=The%20Hough%20transform%20is%20a,shapes%20by%20a%20voting%20procedure. (дата обр. 11.01.2024).

[5] *Line Segment Detector*. URL: https://www.ipol.im/pub/art/2012/gjmr-lsd/article.pdf (дата обр. 2012).

[6] *Оттенки серого*. URL: https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D1%82%D0%B5%D0%BD%D0%BA%D0%B8_%D1%81%D0%B5%D1%80%D0%BE%D0%B3%D0%BE (дата обр. 2023).

[7] *Pattern based corner detection algorithms*. URL: https://research-repository.griffith.edu.au/bitstream/handle/10072/29961/56348_1.pdf;sequence=1 (дата обр. 2009).

[8] *Документ в перспективе. Что делать?* URL: https://habr.com/ru/articles/223507/ (дата обр. 21.05.2014).

[9] *Автоматическое выделение документа на изображении в программе ABBYY FineScanner*. URL: https://habr.com/ru/companies/contentai/articles/200448/ (дата обр. 06.11.2013).